

```

## Events called by the Rdynamic event scheduler (ev$doEvent)

## Main function code

## MUST CHANGE FUNCTION NAME, nstate, ndisc, nextra, Outnames,
## ode CALL, AND structure STATEMENT AT END

vliver_pbpbk <- function(Times, newParms, method="lsode",...){
  Idots <- list(...)
  # How many intervals to solve before giving up:
  maxinter <- 100
  # Number of state variables:
  nstate <- 9
  ndisc <- 0
  nextra <- 1
  #O Names of extra state variables:
  Outnames <- c("Ratioblood2plasma")
  ## Initialize parameters and state variables
 Parms <- initparms(newParms,vliver_pbpbk.Parms,ComputeParms)
yinit <- Inity(Parms)
y <- yinit[[1]]
ydisc <- yinit[[2]]

## put ydisc at the bottom ofParms
Parms <- c(Parms,ydisc)
## initialize the events structure
ev <- ConstructEvents(sort(Times))
## insert dietary inputs
## first set up the pulse
if (any(Parms[["DietInput"]][,2]>0))
{
  ev$newPulse("DietInput")
  # Crop doses that occur after the requested time interval:
  if (!is.null(dim(Parms[["DietInput"]])))
  {
    i <- 1
    while (i <= dim(Parms[["DietInput"]])[1])
    {
      if (max(Times) > Parms[["DietInput"]][i,1]) i <- i + 1
      else break
    }
    Parms[["DietInput"]] <- matrix(Parms[["DietInput"]][1:(i-1),],ncol=2)
    if (Parms[["DietInput"]][i-1,2]!=0)
      Parms[["DietInput"]] <- rbind(Parms[["DietInput"]], matrix(c(max(Times),0),ncol=2))
  } else {
    Parms[["DietInput"]] <- rbind(Parms[["DietInput"]], matrix(c(max(Times),0),ncol=2))
  }
  ## Some code for timing
  ##TimedoEvents <- TimeNextInterval <- Timeinit <- 0
  ## insert all the events for this pulse
  ##ptm <- proc.time()[1]
  if ((zz <- nrow(Parms[["DietInput"]])) > 0) {
    ev$insertEvent(Parms[["DietInput"]][,1],diet_input,-1)
  }
}
##Timeinit <- Timeinit + proc.time()[1] - ptm

```

```

##writeLines(paste("Time to initialize DietInput:", Timeinit,"seconds"))
## update 'now'
now <- Times[1]
#
## setup the matrix to receive the output
ltmp <- list()
inter <- 1
## run through the simulations.
repeat {
  if (ev$quit(now)) break
  ##ptm <- proc.time()[1]
  ys <- ev$doEvents(now,Parms,y,ydisc,ev)
  ##TimedoEvents <- TimedoEvents + proc.time()[1] - ptm
  y <- ys[[1]]
  ydisc <- ys[[2]]
 Parms[names(ydisc)] <- ydisc
  ##ptm <- proc.time()[1]
  nextint <- ev$getNextInterval(now)
  ##if (getOption("Print.nextint")) print(nextint)
  ##TimeNextInterval <- TimeNextInterval + proc.time()[1] - ptm
  ltmp[[inter]] <- ode(y, nextint, "vliver_pbpbk_derivs",Parms, method=method,
    dllname="vLiverPBPK",
    initfunc="vliver_pbpbk_init",
    nout=nextra,outnames=Outnames,...)
  y <- ltmp[[inter]][zz <- nrow(ltmp[[inter]]),2:(nstate+1)]
  now <- ltmp[[inter]][zz,1]
  if (inter > maxinter)
  {
    if (now > max(Times)/10) maxinter <- maxinter*10
    else stop(paste("Solutions only obtained through",now,"after",maxinter,"attempts."))
  }
  inter <- inter + 1
}
tmp <- do.call("rbind",ltmp)
tmp <- tmp[!duplicated(tmp[,1]),]
##writeLines(c(paste("Timing for doEvents:",TimedoEvents),
##           paste("Timing for NextInterval:",TimeNextInterval)))
structure(list(result=tmp[tmp[,1] %in% Times,],
  parameters=Parms,
  model="vliver_pbpbk",
  method=method,
  control=ldots),
  class="RDynOut")
}

vliver_pbpbk_deriv<- function(t,y,newParms, ...){
  ldots <- list(...)
  # Number of state variables:
  nstate <- 9
  ndisc <- 0
  nextra <- 1
  Outnames <- c("Ratioblood2plasma")
  ## Initialize parameters and state variables
 Parms <- initparms(newParms,vliver_pbpbk.Parms,ComputeParms)

```

```

return(DLLfunc("vliver_pbpbk_derivs",0,y,Parms,
  dllname="vLiverPBPK",
  initfunc="vliver_pbpbk_init",
  nout=nextra,outnames=Outnames,...))
}

DLLfunc <- function (func, times, y, parms, dllname, initfunc = dllname,
  rpar = NULL, ipar = NULL, nout = 0, outnames = NULL, forcings = NULL,
  initforc = NULL, fcontrol = NULL)
{
  if (!is.numeric(y))
    stop(`y` must be numeric)
  n <- length(y)
  if (!is.null(times) && !is.numeric(times))
    stop(`times' must be NULL or numeric")
  if (!is.null(outnames))
    if (length(outnames) != nout)
      stop(`length outnames should be = nout")
  ModellInit <- NULL
  Outinit <- NULL
  flist <- list(fmat = 0, tmat = 0, imat = 0, ModelForc = NULL)
  Ynames <- attr(y, "names")
  if (is.null(dllname) || !is.character(dllname))
    stop(`dllname' must be a name referring to a dll")
  if (!is.null(initfunc))
    if (is.loaded(initfunc, PACKAGE = dllname, type = ""))
      is.loaded(initfunc, PACKAGE = dllname, type = "Fortran"))
      ModellInit <- getNativeSymbolInfo(initfunc, PACKAGE = dllname)$address
    }
    else if (initfunc != dllname && !is.null(initfunc))
      stop(paste("cannot integrate: initfunc not loaded ",
        initfunc))
  if (is.null(initfunc))
    initfunc <- NA
  if (!is.null(forcings))
    flist <- checkforcings(forcings, times, dllname, initforc,
      TRUE, fcontrol)
  if (!is.character(func))
    stop(`func' must be a *name* referring to a function in a dll")
  if (is.loaded(func, PACKAGE = dllname))
    Func <- getNativeSymbolInfo(func, PACKAGE = dllname)$address
  }
  else stop(paste("cannot run DLLfunc: dyn function not loaded: ",
    func))
  dy <- rep(0, n)
  storage.mode(y) <- storage.mode(dy) <- "double"
  out <- .Call("call_DLL", y, dy, as.double(times[1]), Func,
    ModellInit, parms, as.integer(nout), as.double(rpar),
    as.integer(ipar), as.integer(1), flist, PACKAGE = "deSolve")
  vout <- if (nout > 0)
    out[(n + 1):(n + nout)]
  else NA
  out <- list(dy = out[1:n], var = vout)
  if (!is.null(Ynames))
    names(out$dy) <- Ynames
  if (!is.null(outnames))

```

```
    names(out$var) <- outnames
    return(out)
}
```